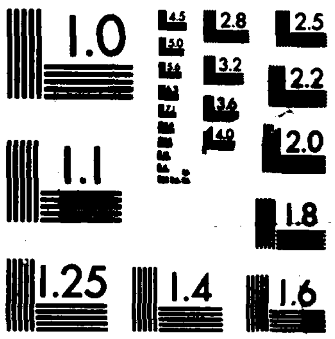
AD-A183 216   METHODOLOGIES FOR CONCURRENT PROGRAMMING(U) TEXAS UNIV   1/1
AT ARLINGTON DEPT OF COMPUTER SCIENCE J MISRA
28 FEB 87 N00014-86-K-0182

UNCLASSIFIED   F/G 12/5   NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

(12)

OFFICE OF NAVAL RESEARCH
FINAL TECHNICAL REPORT

for

March 1, 1986 through February 28, 1987

for
CONTRACT N00014-86-K-0182
TASK NO. NR 4113625

METHODOLOGIES FOR CONCURRENT PROGRAMMING

PRINCIPAL INVESTIGATOR: Jayadev Misra
Computer Sciences Department
The University of Texas at Austin
Austin, Texas 78712

Chapter 1

Parallelism and Programming: A Perspective

Chapter Contents

1. The Unity of the Programming Task

This book is about parallel programs; however, this book is primarily about programs and secondarily about parallelism. The diversity of architectures and consequent programming constructs (send and receive, await, fork and join...) must be placed in the proper perspective with respect to the unity of the programming task. By stressing the differences, we are in danger of losing sight of the similarities. The central thesis of this book is that the unity of the programming task is of primary importance; the diversity is secondary.

The basic problem in programming is managing complexity. We cannot address that problem as long as we lump together concerns about the core problem to be solved, the language in which the program is to be written, and the hardware on which the program is to execute. Program development should begin by focusing attention on the problem to be solved and postponing considerations of architecture and language constructs.

Some argue that in cases where language and hardware are specified as part of a problem, concerns about the core problem, language, and hardware are inseparable. For instance, programs executing on a distributed network of computers must employ some form of message passing; in such cases concerns about message passing appear inseparable from concerns about the core problem. Similarly, since the presence or absence of primitives for process creation and termination in the programming language influence the program, it appears that language issues are inseparable from others. Despite these arguments, we maintain that it is not only possible but important to separate these concerns; indeed it is even more important to do so for parallel programs because parallel programs are less well understood than sequential programs.

Twenty-five years ago, many programs were designed to make optimum use of some specific feature of the hardware. Programs were written to exploit a particular machine language command or the number of bits in a computer word. Now, we know that such optimizations are best left to the last stages of program design or left out altogether. Today, parallel programs are designed much like sequential programs were designed in the 1950's: to exploit the message passing primitives of a language or the network interconnection structure of an architecture. A quarter-century of experience tells us that such optimizations are *best postponed* until the very end of program development. We now know that a physicist who wishes to use the computer to study some phenomenon in plasma physics, for instance, should not begin by asking whether communicating sequential processes or shared-memory is to be used, any more than whether the word size is 32 or 60 bits. Such questions have their place, but concerns must be separated. The first concern is to design a solution to the problem; the later concern is to implement the solution in a given language on a particular architecture. Issues of performance on a specific architecture should be considered, but only at the appropriate time.

Programs outlive the architectures for which they were designed initially. A program designed for one machine will be called upon to execute efficiently on quite dissimilar architectures. If program designs are tightly coupled to the machines of today, program modifications for future architectures will be expensive. Experience suggests that we should anticipate requests to modify our programs to keep pace with modifications in architecture—witness attempts to parallelize sequential programs. It is prudent to design a program for a flexible abstract model of a computer with the intent of tailoring the program to suit future architectures.

An approach to exploiting new architectural features is to add features to the computational model. However, a baroque abstract model of a computer only adds to the complexity of programming. On the other hand, simple models such as the Turing Machine do not provide the expressive power needed for program development. What we desire is a model that is simple and has the expressive power necessary to permit the refinement of specifications and programs to suit target architectures.

The emphasis on the unity of the programming task is a departure from the current view of programming. Currently, programming is fragmented into subdisciplines, one for each architectural form. Asynchronous distributed computing, in which component processes interact by messages, is considered irrelevant to synchronous parallel computing. Systolic arrays are viewed as hardware devices and, hence, traditional ideas of program development are deemed inapplicable to their design.

The goal of this book is to show how programs may be developed in a systematic manner for a variety of architectures and applications. A criticism of this book is that its fundamental premise is wrong because programmers should *not* be concerned with architecture—compilers should. Some styles of programming—e.g., functional and logic programming—are preferred precisely because architecture is not their concern. Our response to this criticism is twofold. First, programmers who are not concerned with architecture should not have to concern themselves with it—they should stop early in the program development process with a program which may or may not map efficiently to the target architecture. Second, there are some problems in which programmers have to be concerned with architecture either because the problem specifies the architecture (e.g., the design of a distributed command and control system) or because performance is critical; for these problems the refinement process is continued until efficient programs for the target architectures are obtained.

2. A Search for a Foundation of Parallel Programming

We seek a small theory that is applicable to programming for a wide range of architectures and applications. The issues that we consider central to such a theory are: nondeterminism, absence of control flow, synchrony/asynchrony, states and assignments, proof systems that support program development by stepwise refinement of specifications, and the decoupling of correctness from complexity, i.e., of programs from architectures. These issues are elaborated on next.

2.1 Nondeterminism

How can we develop programs for a variety of architectures through a series of refinements? *By specifying program execution at an appropriate level of detail: by specifying little in the early stages of design, and by specifying enough in the final stages to ensure efficient executions on target architectures.* Specifying little about program execution means that our programs may be nondeterministic. Different runs of the same program may execute statements in different orders, consume different amounts of resources and even produce different results.

Nondeterminism is useful in two ways. First, nondeterminism is employed to derive simple programs, where simplicity is achieved by avoiding unnecessary determinism;

such programs may be optimized by limiting the nondeterminism, i.e., by disallowing executions unsuitable for a given architecture. Second, some systems, e.g., operating systems and delay-insensitive circuits, are inherently nondeterministic; programs that represent such systems have to employ some nondeterministic constructs.

2.2 Absence of Control Flow

The notion of sequential control flow is pervasive in computing. Turing Machines and von Neumann computers are examples of sequential devices. Flow charts and early programming languages were based on sequential flow of control. Structured programming retained sequential flow of control and advocated problem decomposition based on sequencing of tasks. The prominence of sequential control flow is partly due to historical reasons. Early computing devices and programs were understood by simulating their executions sequentially. Many of the things we use daily, such as recipes and instructions for filling out forms, are sequential; this may have influenced programming languages and the abstractions used in program design.

The introduction of co-routines was an indication that some programs are better understood through abstractions that are not related to control flow. A program structured as a set of processes is a further refinement: it admits multiple sequential flows of control. However, processes are viewed as *sequential* entities—note the titles of two classic papers in this area, "Cooperating Sequential Processes" in Dijkstra [1968] and "Communicating Sequential Processes" in Hoare [1978]. This suggests that sequential programming is the norm, and parallelism, the exception.

Control flow is not a unifying concept. Programs for different architectures employ different forms of control flow. Program design at early stages should not be based on considerations of control flow; it is a later concern. It is easier to restrict flow of control in a program having few restrictions than to remove unnecessary restrictions from a program having too many.

The issue of control flow has clouded several issues. Let us review one. Modularity is generally accepted as a Good Thing. What is a module? A module implements a set of related concerns, it has clean narrow interfaces, and the states of a system when control flows into and out of the module are specified succinctly. Now, a clean, narrow interface is one issue and control flow into and out of a module is another. Why not separate them? In our program model, we retain the concept of module as a part of a program that implements a set of related concerns. Yet, we have no notion of control flow into and out of a module. Divorcing control flow from module construction results in an unconventional view of modules and programming—though a useful one, we believe, for the development of parallel programs.

2.3 Synchrony and Asynchrony

Synchronous and asynchronous events are at the core of any unified theory of parallel programming. For instance, all events in a systolic array are synchronous: at each clock tick all processors in the array carry out a computational step. On the other hand, a data network spanning the globe has no common clock; processes at different nodes of the network execute steps asynchronously. Some systems have synchronous components interconnected by asynchronous channels—an example of such a system is an electronic circuit consisting of synchronous subcircuits interconnected by wires with arbitrary delays. Partitioning systems into synchronous and asynchronous varieties is artificial; a theory should include synchrony and asynchrony as fundamental concepts.

2.4 States and Assignments

A formal model employed by computing scientists, control theorists, communication engineers, circuit designers, and operations researchers (among others) is the state transition system. Computing scientists use state transition models in studying formal languages. Control theorists represent the systems they study as continuous or discrete state transition models—a typical control problem is to determine an optimal trajectory in a state space. Markov processes, employed by communication engineers and operations researchers, are state transition systems. Communication engineers represent communication protocols as state transition systems. Physical systems are often described in terms of state transitions. Therefore, it appears reasonable to us to propose a unifying theory of parallel programming based on state transition systems; indeed, it is our hope that the theory will be helpful to engineers and natural scientists as well as to programmers. However, treating a program as a state transition system—a set of states, an initial state, and a state transition function—offers little for a methodology of program development. Too much of the semantics of a problem is lost when it is represented as a set of states and transitions. Therefore, we wish to employ the theory of state transition systems while enjoying the representational advantages of programming languages. One way of doing so is to employ variables and assignments in the notation.

A starting point for the study of assignments is the following quote from Backus [1978]:

> "... the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.
>
> The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements of the language exist in order to make it possible to

perform a computation that must be based on this primitive construct: the assignment statement.

This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control."

One cannot but agree that disorderly programming constructs are harmful. But there *is* an orderly world of assignments. The problems of imperative programming may be avoided while retaining assignments.

Word-at-a-time bottleneck: Multiple assignments allow assignments to several variables simultaneously; these variables may themselves be complex structures.

Control flow: Assignment can be divorced from control flow. We propose a program model that has assignments but no control flow.

Mathematical properties: A program model based on assignments and without control flow has remarkably nice properties.

2.5 Extricating Proofs from Program Texts

One way of proving the correctness of a sequential program is to provide an annotation of it; the proof consists of demonstrating that a predicate holds at a point in the text of the program—thus the proof is inextricably intertwined with the program text. We seek a proof system that allows the proof to be extricated from the program text. This would allow us to develop and study a proof in its own right.

Much of program development in our methodology consists of refining specifications, i.e., adding detail to specifications. Given a problem specification, we begin by proposing a general solution strategy. Usually the strategy is broad; it admits many solutions. Next we give a specification of the solution strategy and prove that the solution strategy (as specified) solves the problem (as specified). When we consider a specific set of target architectures, we may choose to narrow the solution strategy, which means refining the specification further. At each stage of strategy refinement, the programmer is obliged to prove that the specification proposed is indeed a refinement of a specification proposed at an earlier step. The construction of a program is begun only after the program has been specified in extensive detail. Usually, the proof that a program fits the detailed specification is straightforward because much of the work associated with proofs is carried out in earlier stages of stepwise refinement. We seek methods of specification and proof that do not require a skeleton of the program text to be proposed until the final stages of design. This is a departure from conventional sequential program development, where it is quite common to propose a skeleton of the program text early in the design, and where refinement of a specification proceeds hand-in-hand with the addition of flesh to the program skeleton.

2.6 Separation of Concerns: Correctness and Complexity

A point of departure of our work from the conventional view of programming is this: We attempt to decouple a program from its implementation. A program may be implemented in many different ways—a program may be implemented on different architectures, and even for a given computer, a program may be executed according to different schedules. The correctness of a program is independent of the target architecture and the manner in which the program is executed; by contrast, the efficiency of a program execution depends on the architecture and manner of execution. Therefore, we do not associate complexity measures with a program but rather with a program *and a mapping* to a target computer. A mapping is a description of how programs are to be executed on the target machine and a set of rules for computing complexity measures for programs when executed on the given target machine. A programmer's task, given a specification and a target architecture, is to derive a program with its proof, select a mapping that maps programs to the given target architecture, and then evaluate complexity measures.

The operational model of a program—how a computer executes a program—is straightforward for programs written in conventional sequential imperative languages, such as PASCAL, executing on conventional sequential machines. Indeed, many sequential imperative languages (so-called von Neumann languages) have been designed so that the manner in which von Neumann machines execute programs, written in these languages, is self-evident. The complexity measures (i.e., metrics of efficiency) of a program written in such a language are the amounts of resources, such as time and memory, required to execute the program on a von Neumann architecture. Usually, when computing scientists refer to complexity measures of a program, they implicitly assume a specific operational model of a specific architecture—in most cases the architecture is the traditional, sequential architecture and its abstract model is the Random Access Machine or RAM.

The tradition of tightly coupling programming notation to architecture, inherited from von Neumann languages and architectures, has been adopted in parallel programming as well. For instance, programmers writing in Communicating Sequential Processes (CSP) notation usually have a specific architecture in mind, *viz.* an architecture that consists of a set of von Neumann computers that communicate by means of message passing.

In this book, the correctness of programs (in Chapter 3) is treated as a topic separate from architectures and mappings (in Chapter 4). A brief discussion of mappings is included in the following section.

3. Introduction to the Theory

In this book, we introduce a theory—a computational model and a proof system—called UNITY. We choose to view our programs as Unbounded Nondeterministic Iterative Transformations—hence the term UNITY. In the interest of brevity, the phrase "a UNITY program" is preferred to "a program in unbounded nondeterministic iterative transformation notation." We are not proposing a programming language. We adopt the minimum notational machinery to illustrate our ideas about programming. This section is (even as introductions go) incomplete. A thorough description of notation and proof rules appears in the next two chapters.

3.1 UNITY Programs

A program consists of a declaration of its variables, a specification of their initial values, and a set of multiple assignment statements. A program execution starts from any state satisfying the initial condition and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following "fairness" rule: every statement is selected infinitely often.

Our model of programs is simple; in fact it may appear too simple for effective programming. We show in this book that our model is adequate for the development of programs in general and parallel programs in particular. Now, we give an informal and very incomplete description of how this model addresses some of the issues described in section 2.

3.2 Separating Concerns: Programs and Implementations

A UNITY program describes *what* should be done in the sense that it specifies what the initial state and the state transformations (i.e., the assignments) are. A UNITY program does not specify precisely *when* an assignment should be executed—the only restriction is a rather weak fairness constraint: every assignment is executed infinitely often. Nor does a UNITY program specify *where*, i.e., on which processor in a multiprocessor system, an assignment is to be executed, or to which process an assignment belongs. Also, a UNITY program does not specify *how* assignments are to be executed or *how* an implementation may halt a program execution.

UNITY separates concerns between *what* on the one hand and *when*, *where* and *how* on the other. The *what* is specified in a program, whereas the *when*, *where* and *how* are specified in a mapping. By separating concerns in this way, a simple programming notation is obtained that is appropriate for a wide variety of architectures. Of course, this simplicity is achieved at the expense of making mappings immensely more important and more complex than they are now.

3.3 Mapping Programs to Architectures

In this section, we give a brief outline of mappings of UNITY programs to several architectures. We consider the von Neumann architecture, synchronous shared-memory multiprocessors, and asynchronous shared-memory multiprocessors. The description given here is sufficient for understanding how the example programs of the next section are to be executed on various architectures. The subject of mapping is treated in more detail in Chapter 4. Though we describe mappings from UNITY programs to architectures, UNITY programs can also be mapped to programs in conventional programming languages.

A mapping to a von Neumann machine specifies the schedule for executing assignments and the manner in which a program execution terminates. The implementation of multiple assignments on sequential machines is straightforward and is not discussed here. We propose a mapping in which an execution schedule is represented by a finite sequential list of assignments in which each assignment in the program appears at least once. The computer executes this list of assignments repeatedly forever (but, see below). We are obliged to prove that the schedule is fair, i.e., that every assignment in the program is executed infinitely often. Since every assignment in the program appears at least once in the list, and since the list is executed forever, it follows that every assignment in the program is executed infinitely often.

Given that a UNITY program execution does not terminate, how do we represent traditional programs whose executions do terminate (in the traditional sense)? We regard termination as a feature of an implementation. A cleaner theory is obtained by distinguishing program execution—an infinite sequence of statement executions—from its implementation—a finite prefix of the sequence.

A state of a program is called a *fixed point* if and only if execution of any statement of the program, in this state, leaves the state unchanged. A predicate, called FP (for fixed point), characterizes the fixed points of a program. It is the conjunction of the equations that are obtained by replacing the assignment operator by equality in each of the statements in the program. Therefore, FP holds if and only if values on left and right sides of each assignment in the program are identical. Once FP holds, continued execution leaves values of all variables unchanged, and therefore, it makes no difference whether the execution continues or terminates. One way of implementing a program is to halt the program after it reaches a fixed point.

A *stable predicate* or *stable property* of a program is a predicate that continues to hold, once it holds. Thus, FP is a stable property. The detection of fixed points is treated in Chapter 9, and the detection of general stable properties in Chapters 10 and 11.

In a synchronous shared-memory system, a fixed number of identical processors share a common memory which can be read and written by any processor. There is a common clock, where in each clock tick, every processor carries out precisely one step of computation. The synchrony inherent in a multiple assignment statement makes it convenient to map such a statement to this architecture: each processor computes the expression on the right side of the assignment corresponding to one variable and then assigns the computed value to this variable. This architecture is also useful for computing the value of an expression which is defined by an associative operator, such as sum, minimum or maximum, applied to a sequence of data items. Details are given in Chapter 4.

An asynchronous shared-memory multiprocessor consists of a fixed set of processors and a common memory, but there is no common clock. If two processors access the same memory location simultaneously, then their accesses are made in some arbitrary order. A UNITY program can be mapped to such an architecture by partitioning the statements of the program among the processors. In addition, a schedule of execution for each processor should be specified that guarantees fairness of execution for the partition. Observe that if execution for every partition is fair, then any fair interleaving of these executions determines a fair execution of the entire program. Our suggested mapping assumes a coarse grain of atomicity in the architecture: two statements are not executed concurrently if one modifies a variable that the other one uses. Hence, the effect of multiple processor execution is the same as a fair interleaving of their individual executions. Mappings under finer grains of atomicity are considered in Chapter 4. In that chapter we also show that an asynchronous multiprocessor can simulate a synchronous multiprocessor by simulating the common clock; hence, programs for the latter can be executed on the former.

To evaluate the efficiency of a program executed according to a given mapping, it is necessary to describe the mapping—the data structures and the computational steps—in detail. Descriptions of architectures and mappings can be made extremely detailed. Memory caches, I/O devices and controllers can be described if it is necessary to evaluate efficiency at that level of detail, but we shall not do so in this book because we merely wish to emphasize the separation of concerns: programs are concerned with *what* is to be done whereas mappings are concerned with the implementation details of *where, when,* and *how.*

3.4 Modeling Conventional Programming Language Constructs

In this section, we show that conventional programming language constructs that exploit different kinds of parallelism have simple counterparts in UNITY. This is not too surprising because the UNITY model incorporates both synchrony—a multiple assignment assigns to several variables synchronously—and asynchrony—nondeterministic selection leaves unspecified the order of executions of statements.

A synchronous system is one in which there is a global clock variable that is incremented with every state change. Multiple assignments model parallel synchronous operations.

A statement of the form **await** B **do** S in an asynchronous shared-variable program is encoded as a statement in our model which does not change the value of any variable if B is *false* and otherwise has the same effect as S. A Petri net, another form of asynchronous system, can be represented by a program in which a variable corresponds to a *place*, the value of a variable is the number of *markers* in the corresponding place, and a statement corresponds to a *transition*. The execution of a statement decreases values of variables corresponding to its input places by 1 (provided they are all positive) and increases values of variables corresponding to its output places by 1 in one multiple assignment.

Asynchronous message-passing systems with first-in-first-out error-free channels may be represented by encoding each channel as a variable whose value is a sequence of messages (representing the sequence of messages in transit along the channel). Sending a message is equivalent to appending the message to the end of the sequence and receiving a message to removing the head of the sequence.

We cannot control the sequence in which statements are executed. However, by using variables appropriately in conditional expressions, we can ensure that the execution of a statement has no effect (i.e., does not change the program state) unless the statement execution occurs in a desired sequence.

4. An Example: Scheduling a Meeting

The goal of this example is to give the reader *some* idea of how we propose to develop programs. Since the theory is presented only in the later chapters, the discussion here is incomplete. The thesis of this book is unusual and the computational model even more so; skeptical readers may want to get a rough idea of how we propose to design programs before investing their time any further—this example is to satisfy their need.

4.1 The Problem Statement

The problem is to find the earliest meeting time acceptable to every member of a group of people. Time is integer-valued and nonnegative. To keep notation simple, assume that the group consists of three people called F, G, and H. Associated with persons F, G, H are functions f, g, h (respectively) that map times to times. The meaning of f is as follows (and the meanings of g, h follow by analogy). For any t, $f(t) \geq t$; person F can meet at time $f(t)$ and cannot meet at any time u where $t \leq u < f(t)$. Thus, $f(t)$ is the earliest time at or after t at which person F can meet. (Note: From the problem description, f is a monotone nondecreasing function of its argument and $f(f(t)) = f(t)$. Also, $t = f(t)$ means that F can meet at t.) Assume that there exists some common meeting time z. In the interest of brevity we introduce a boolean function *com* (for *com*mon meeting time) over nonnegative integers defined as follows:

$$com(t) \equiv [t = f(t) = g(t) = h(t)]$$

Problem Specification

Note: All variables r, t, z, referred to in the specification, are nonnegative integers.

Given integer-valued functions f, g, h, where for all t :

$$f(t) \geq t \ \wedge \ g(t) \geq t \ \wedge \ h(t) \geq t \ \wedge \qquad \{f, g, h \text{ are monotone nondecreasing}\}$$

$$f(f(t)) = f(t) \ \wedge \ g(g(t)) = g(t) \ \wedge \ h(h(t)) = h(t)$$

and given a z such that $com(z)$, design a program that has the following as a stable predicate:

$$r = \min\{t \mid com(t)\}$$

Furthermore, the program establishes this stable predicate within a finite number of steps of execution.

Discussion

There are many ways of attacking this problem. One approach is to structure the solution around a set of processes, one process corresponding to each person; the behavior of people provides guidelines for programming these processes. We propose an alternate approach based on our theory. We describe both methods as applied to this problem starting with the operational view. The discussion is, perforce, incomplete because a thorough description of the theory is presented only in later chapters.

4.2 Operational, Process-Oriented Viewpoint

Here we describe only person F's behavior; the behaviors of G and H follow by analogy. Consider persons seated at a round table and a letter containing a proposed meeting time (initially 0), passed around among them. Person F, upon receiving the letter with time t, sets the proposed time to $f(t)$ and passes the letter to the next person. If the letter makes one complete round without a change in the proposed meeting time then this time is the solution.

Another strategy is to use a central coordinator to whom each person reports the next time at which he can meet. The coordinator broadcasts t, the maximum of these times, to all persons and F then sends $f(t)$ back to the coordinator. These steps are repeated until the coordinator receives identical values from all persons.

Yet another solution is to divide the persons into two groups and recursively find meeting times for each group. Then the maximum of these times is used as the next estimate for repeating these steps, unless the two values are equal.

Another approach is to use a bidding scheme where an auctioneer calls out a proposed meeting time t, starting at $t = 0$, and F can raise the bid to $f(t)$ (provided this value exceeds t). The common meeting time is the final bid value, i.e., a value that can be raised no further. The reader may develop many other solutions by casting the problem in a real-world context. Different solutions are appropriate for different architectures.

4.3 The UNITY Viewpoint

We take the specification as the starting point for program design. A number of heuristics are given in later chapters for constructing programs from specifications. For this *example*, we will merely propose programs and argue that they meet their specifications. Our next concern, after designing a program, is to refine it further so that it can be mapped to a target architecture for efficient execution. We may have to consider alternate refinements of the same program, or even alternate programs, if we are unable to find a mapping with the desired efficiency.

A Simple Program

The problem specification suggests immediately the following program. The syntax is as follows: the assignment statements are given under **assign**. Declarations of variables have been omitted.

Program P1

assign $r := \min\{u \mid (0 \le u \le z) \wedge (com(u))\}$

end {**P1**}

This program's correctness needs no argument. Now, we consider how best to implement P1 on different architectures.

For a von Neumann machine, computation of the right side of the assignment—i.e., finding the first u for which $com(u)$ holds—can proceed by checking the various values of u, from 0 to z, successively. The program is entirely straightforward. The number of steps of execution is proportional to the value of r; in the worst case, it is $O(z)$.

For a parallel synchronous machine (see Chapter 4) the minimum over a set of size z can be computed in $O(log\ z)$ steps by $O(z)$ processors and, in general, in $O(z/k + log\ k)$ steps by $O(k)$ processors.

For parallel asynchronous shared-memory systems, a similar strategy can be employed for the computation of the minimum.

Discussion

We showed a very simple program, the correctness of which is obvious from the specification. We described, informally, how the program could be mapped to different architectures. It is possible to refine this program so that the mappings correspond more directly to its statements and variables. For instance, to describe the mapping to an asynchronous message-passing system, we may refine this program by introducing variables to represent communication channels and statements to simulate sending and receiving from these channels; then, we can describe the mapping by specifying which statements are to be executed by which processors and which channels connect which pairs of processors.

We make some general observations about Program P1 independent of the architecture on which it is implemented. We note that $com(u)$ has to be evaluated for *every* u, $0 \le u \le z$. This may be wasteful because from the problem statement we can deduce that no u can be a common meeting time—i.e., $com(u)$ does not hold—if $t \le u < f(t)$. Therefore, it is not necessary to evaluate $com(u)$ for any u in this interval. Such an approach is taken in the next program.

Another Simple Program

The program that we propose is so straightforward that we give it without a prior detailed discussion. The symbol $[\!]$ is used to separate the assignment statements in the following program; the initial condition is specified under **initially**.

Program P2

```
  initially  $r = 0$

  assign     $r := f(r) \;[\!]\; r := g(r) \;[\!]\; r := h(r)$

end {P2}
```

The program has three assignments: $r := f(r)$, $r := g(r)$, and $r := h(r)$. Computation proceeds by executing any one of the three assignments selected nondeterministically. The selection obeys the fairness rule: every assignment is executed infinitely often.

This program may be understood as follows. Initially, the proposed meeting time is zero. Any one of the participants—F, G, or H—increases the value of the proposed meeting time, if he cannot meet at that time, to the next time at which he can meet; in this sense, this program is similar to the bidding scheme outlined in section 4.2. At fixed point, r is a common meeting time.

Proof of Correctness

In the absence of a proof theory (which is developed in Chapter 3), the best we can do is to sketch an informal proof. The proof, given next, can be made more concise by employing the formalisms given later in the book. We divide the argument into three parts.

(1) We claim that the following predicate is true at all points during program execution; such a predicate is called an invariant.

invariant $(0 \leq r) \wedge \langle \text{for all } u \text{ where } 0 \leq u < r :: \neg com(u)\rangle$

In words, the invariant says that r is nonnegative and that there is no common meeting time earlier than r. Using the specification and this invariant, it is seen that $r \leq z$ is always true, because there is a common meeting time at z.

To prove the invariant, we show that it is true initially and execution of any statement preserves its truth. Initially, $r = 0$. Hence, the first conjunct in the invariant is true and the second conjunct holds, vacuously. Now consider execution of the statement $r := f(r)$. We know before execution of this statement, from definition, that F cannot meet at any u, $r \leq u < f(r)$. Hence, $\neg com(u)$ holds for all u, $r \leq u < f(r)$. Also, from the invariant, we may assume that $\neg com(u)$ holds for all u, $0 \leq u < r$. Trivially, $0 \leq f(r)$. Therefore,

$$(0 \leq f(r)) \wedge \langle \text{for all } u \text{ where } 0 \leq u < f(r) :: \neg com(u)\rangle$$

holds prior to execution of $r := f(r)$. The effect of execution of this statement is to set r to $f(r)$. Replacing $f(r)$ by r in the above predicate, we see that the invariant continues to hold after execution of this statement.

Due to symmetry among the statements, similar arguments show that the invariant is preserved by executing any statement.

(2) From the definition, FP for this program (which is the conjunction of the equations obtained by replacing := by = in every statement) is

$$\text{FP} \equiv r = f(r) \wedge r = g(r) \wedge r = h(r) .$$

From the definition of $com(r)$ it follows that $\text{FP} \equiv com(r)$.

Combining the results proved in parts (1) and (2), we claim that if Program P2's execution reaches a fixed point then the value of r is the earliest meeting time. Our remaining task is to show that every execution of Program P2 does, indeed, reach a fixed point; this is shown below.

(3) We show that if $\neg FP \wedge r = k$ holds at any point during computation then $r > k$ holds at some later point. Thus, r keeps on increasing as long as $\neg FP$ holds. We showed in part (1) that r cannot increase beyond z. Therefore, eventually FP holds. The proof of the claim that r increases if $\neg FP$ holds, is as follows. From the FP given in part (2),

$$\neg FP \wedge (r = k) \equiv k < f(k) \vee k < g(k) \vee k < h(k)$$

Suppose $k < f(k)$ (similar reasoning applies for the other cases). From the fairness requirement, the statement $r := f(r)$ is executed some time later. Since the value of r never decreases, just prior to execution of this statement, $r \geq k$ (r may have increased in the meantime) and hence,

$f(r) \geq f(k) > k$.

The effect of the execution of the statement is to set r to $f(r)$, and hence r increases beyond k.

This completes the proof.

Mapping Program P2 to Various Architectures

We propose a mapping from Program P2 to a von Neumann machine. The mapping is described by a list of assignments (see section 3.3). The list we propose is:

$$r := f(r) \; ; \; r := g(r) \; ; \; r := h(r)$$

This list of assignments is executed repeatedly until a fixed point is detected—a fixed point is detected when three consecutive assignments do not change the value of r. This program corresponds to a token, containing r, being passed in a circular fashion from F to G to H and then back to F, with each person setting r to the next time at or after r at which he can meet. When the token passes by all three persons without being modified, the program terminates.

Now suppose the functions f, g, h are such that it is more efficient to apply f twice as often as g or h. Therefore we wish to repeatedly execute the following cycle: apply f, then g, then f again and then h. So we employ a different mapping from P1 with the execution schedule represented by the following list of assignments:

$$r := f(r) \; ; \; r := g(r) \; ; \; r := f(r) \; ; \; r := h(r)$$

The point of showing two different mappings is to emphasize that P2 describes a family of programs, each one corresponding to a different schedule. By proving P2, we have proved the correctness of all members in the family. Observe that each member of the family can also be represented as a UNITY program. For instance, the first schedule corresponds to a program with a single assignment statement:

$$r := h(g(f(r)))$$

and the second one to a program with the assignment statement:

$$r := h(f(g(f(r)))) \; .$$

Program P2 can be implemented on asynchronous multiprocessors by partitioning its statements among the processors. For instance, we may employ three processors, each executing one statement, and r's value resides in the common memory.

We show yet another member of this family of programs obtained according to a different schedule. As in the first schedule, the three statements are executed in some order in a "pass" and then the pass is repeated. However, the order in which the statements are executed within a pass depends on the value of r before the pass starts.

For definiteness, suppose $f(r) \leq g(r) \leq h(r)$ at the beginning of a pass. Then, the statements are executed in the following order:

$$r := f(r) \; ; \; r := g(r) \; ; \; r := h(r) \; .$$

It is easy to see that the effect of a pass is to set r to the maximum of $f(r)$, $g(r)$, $h(r)$. Hence, we have established the correctness of the following program.

Program P3

```
  initially  $r = 0$

  assign     $r := \max(f(r), g(r), h(r))$

end {P3}
```

This program is similar in spirit to the central coordinator scheme outlined in section 4.4. It is a suitable starting point for programming parallel synchronous multiprocessors. If the number of persons is N ($N = 3$ in this case), the maximum on the right side of the assignment can be computed in $O(log\ N)$ steps using $O(N)$ processors.

Discussion

It is possible to develop yet other programs and show alternate schedules and mappings to various architectures. We have attempted to illustrate only that certain *concerns*, particularly dealing with architectures, may be ignored at higher levels of design, and introduced at lower levels based on considerations of efficiency.

5. Summary

document

This ~~book~~ proposes a unifying theory for the development of programs for a variety of architectures and applications. The computational model is unbounded nondeterministic iterative transformations of the program state. ~~In this book,~~ transformations of the program state are represented by multiple assignments. The theory attempts to decouple the programmer's thinking about a program and its implementation on an architecture; we attempt to separate the concerns of *what* from those of *where, when,* and *how.* Details about implementations are considered in mappings of programs to architectures. We hope to demonstrate that we can develop, specify and refine solution strategies independent of architectures.

The utility of a new approach is suspect, especially when it is a radical departure from the conventional. Therefore, we have made a conscientious effort to apply our ideas to a number of architectures and application domains. Our experience is encouraging.

Keywords: parallel programming; UNITY; computer programming.

Bibliographic Notes

UNITY was first discussed at the Conference on The Principles of Distributed Computing in Vancouver, B.C. in August 1984 by Chandy [1984]. The basic idea has not changed since then. Applications of UNITY in the literature include termination detection algorithms, Chandy and Misra [1986a], systolic programs, Chandy and Misra [1986b], Snepscheut and Swenker [1987], and self stabilizing programs, Brown [1987].

The idea of modeling programs as state-transition systems is not new. Work of Manna and Pnueli [1983], Pnueli [1981], and Lamport and Schneider [1984] on using transition systems to model distributed systems are particularly relevant. Lynch and Tuttle [1987] use transition systems to develop distributed programs by stepwise refinement. A state based model of distributed dynamic programming is in Bertsekas [1982]. A stimulus-response based approach for program specification is proposed in Parker et al. [1980].

The idea of using mappings to implement programs on different architectures is not new either. The work, since the 1960's, of recognizing parallelism in sequential programs is an instance of employing mappings to parallel architectures. The point of departure of UNITY is to offer a different computational model from which to do the mapping. Formal complexity models of architectures are in Aho et al. [1983], Goldschlager [1977], Pippenger [1979], and Cook [1983]. A survey of concurrent programming constructs is in Andrews and Schneider [1983].

The importance of nondeterminism was stressed by Dijkstra [1976], and he proposed the nondeterministic construct—the guarded command. The guarded command is used extensively in Hoare [1984] which contains a definitive treatment of program structuring using Communicating Sequential Processes. A comprehensive treatment of fairness is in Francez [1986]. In its use of nondeterminism, UNITY is similar to expert-system languages such as OPS5, Brownston et al. [1985]. Milner [1983] contains a calculi for synchrony and asynchrony. Finally, we wish to point out that some of the initial motivation for UNITY came from difficulties encountered in using spreadsheets, a notation that has not received much attention from the computing sciences community.

Final Technical Report of

ONR Grant #N00014-86-K-0182

Summary of Work Accomplished

Under this grant we had proposed to study a novel form of programming which includes assignment statement as the only construct. This form of programming was originally suggested by K. Mani Chandy in his keynote address at the Conference on Principles of Distributed Computing, Vancouver, August 1984. Since then, Chandy and I have extensively explored this area. We are close to completion of a book on this subject, *Parallel Program Design: A Foundation* (to be published by Addison-Wesley, January 1988). The work accomplished has far exceeded the proposal and our initial expectations. It has attracted attention of the leading computer scientists, and several groups in the U.S.A. and Europe are already using this form of programming in various application areas.

The major highlights of the programming system are as follows: A program consists of a set of assignment statements. Each assignment statement assigns values to one or more variables (in the latter case, it is known as a multiple assignment statement). The program execution consists of repeating the following steps forever: choosing a statement arbitrarily and executing it. The execution is constrained by the following fairness criterion: in an infinite execution, every statement is selected infinitely often.

The proposal under the ONR Grant was to develop for this kind of programming,

- a theory and,

- methodologies

We elaborate on these two aspects below. Here, we remark that in addition to work in these areas, substantial understanding has been gained in applying the programming technique to various areas of computer science.

The theory for our programming system consists of a logic which is used to make statements about the program states that may arise during execution. Since our interest is in both terminating and non terminating programs, our logic is meant to express properties of both these types of systems. We introduce one operator of temporal logic—*unless*—in our logic and we define two new operators—*ensures* and *leads-to*. The first operator is adequate for stating all safety properties of the systems; the last two are used for stating progress properties. An elaborate theory has been built up around these operators.

The methodology of programming is based on suggesting various ways to compose program components to build a larger program. After extensive study, we have proposed two structuring mechanisms—*union* and *superposition*—that seem to be generally useful for this form of programming. These structuring mechanisms are surprisingly powerful. The union mechanism is adequate for describing asynchronous shared-memory systems and networks of communicating processes. Superposition is the basis underlying program development in layers. We have developed the theories needed to reason about programs which are constructed by applying these structuring mechanisms; using our theory, it is possible to deduce the properties of a constructed program from the properties of its components.

The work promises to be exciting. It shows an entirely novel way of approaching the problem of system design, based on sound formalisms and yet economical in practice.

ONR N00014-86-K-0182

PUBLICATIONS

1. Jayadev Misra (with K. Mani Chandy), "Systolic Algorithms as Programs", **Distributed Computing**, Vol. I, No. 3, August 1986 (177-183).

2. Jayadev Misra (with K. Mani Chandy), "How Processes Learn", **Distributed Computing**, Vol. I, No. 1, January 1986 (40-52).

3. K. Mani Chandy and Jayadev Misra, **Parallel Program Design: A Foundation**, Addison-Wesley, to appear 1988. (The research results reported and acknowledged in this book were partially supported by grants from ONR, AFOSR, and IBM.)

4. Edgar Knapp, "Deadlock Detection in Distributed Databases", submitted to **ACM Computing Surveys**, 1987.

Technical Report

A.Singh, J.Anderson, M.Gouda, "The Elusive Atomic Register Revisited", (TR-86-30), accepted for publication in the **Proceedings of the Sixth ACM SIGACT-SIGOPS Symposium on PODC**, December 1986.

INVITED LECTURES

1. Jayadev Misra, "A Model for Nondeterministic Computations", California Institute of Technology, Pasadena, California, December 12, 1985.

2. Jayadev Misra, three-lecture series on "A Foundation of Programming Based on Nondeterminism", Microelectronics & Computer Technology Corporation (MCC), Austin, Texas, Spring 1986.

3. Jayadev Misra, "Toward a Unified View of Parallel Programming", Carnegie-Mellon University, Yale University, November 9-12, 1986.

GRADUATE STUDENTS SUPPORTED BY THIS GRANT

Mark Staskauskas	Robert Comer
Ambuj Singh	Rajive Bagrodia

END
9-87
DTIC